Operating Systems (2INC0)

Condition Synchronization (07)
Condition variables and signals

Dr. Geoffrey Nelissen
Courtesy of Prof. Dr. Johan Lukkien and
Dr. Tanir Ozcelebi

IRiS
Interconnected
Resource-aware
Intelligent Systems

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Recall: action synchronization

- **Cooperation → Synchronization**
  - N concurrent processes **working for a common goal**
  - **Goal 1**: avoid program traces that may violate a certain invariant
  - **Goal 2**: steer the execution to maintain some property
    - e.g., enforce an execution order
  - Typically, by **blocking** the task execution **until an assertion becomes true**

- **Competition → Mutual exclusion**
  - N concurrent processes **competing for a resource**
  - Tasks define *critical sections* (CS)
  - **Introduce extra synchronization requirements to access the CS.**

Condition synchronization: motivation

- **Action synchronization**
  - the invariant refers to ordering of actions:
  - → enforced by **counting** how many times **actions** is performed

- **Problem:** just **counting may not be enough** to solve all synchronization problems
  - Examples:
    - *x := y+z; or x := 2*y;* Value of a variable may not be inferred by simply counting the actions on that variable
    - wait until *x=0 ∨ y=0 :* condition contains a disjunctions (OR operations)

- **Solution: Condition synchronization**
  - the invariant refers to a (combination of) execution state(s)
  - requires **explicit communication (signalling)** between processes

Condition synchronization: motivation

- Needed **when just counting is not enough** to solve a synchronization problem
- Allows to **simplify** otherwise complex sequences of semaphore operations
  - *P(a); P(a); P(b); P(m); ....*

Two principles:

Where a condition may be violated: **check and block**

Where a condition *may have become true*: **signal waiters**

<u>Condition</u> synchronization: building blocks

- var *cv* : ***Condition***;   (often **associated with** a condition: **a Boolean expression *B* in terms of program variables;**)

- 4 basic operations on variable *cv*:
  - *Wait(..., cv)*   **suspend** execution of caller
  - *Signal(cv)*   **free <u>*one*</u> process**/thread suspended on *Wait(cv)* (void if there is none)
  - *Sigall(cv)*   **free <u>*all*</u> processes** suspended on *Wait(cv)*
  - *Empty(cv)*   Check if "there is no process suspended on *Wait(cv)*" (returns true or false)

- Extra operations in specific implementations.

Example

- **Maintain *x≥0* in a program with arbitrary assignments to *x***

```
Proc P1
|[
  while( true) {
      x := x-10;
  }
]|
```

```
Proc P2
|[
  while( true) {
      x := x + 50;
      x := 2*x;
  }
]|
```

Example

- **Maintain *x≥0* in a program with arbitrary assignments to *x***

```
var cv: condition;
    m: binary semaphore (initially := 1)
```

```
Proc P1
|[
  while( true) {
      x := x-10;
  }
]|
```

```
Proc P2
|[
  while( true) {
      x := x + 50;
      x := 2*x;
  }
]|
```

Example

- **Maintain *x≥0* in a program with arbitrary assignments to *x***

```
var cv: condition;
    m: binary semaphore (initially := 1)
```

Define the **critical sections**

```
Proc P1
|[
  while( true) {
      x := x-10;
  }
]|
```

```
Proc P2
|[
  while( true) {
      x := x + 50;
      x := 2*x;
  }
]|
```

Example

- **Maintain *x≥0* in a program with arbitrary assignments to *x***

var *cv*: ***condition***;
m: ***binary semaphore*** *(initially := 1)*

Define the **critical sections**

```
Proc P1
|[
  while( true) {
      P(m);
      x := x-10;
      V(m);
  }
]|
```

```
Proc P2
|[
  while( true) {
      P(m);
      x := x + 50;
      x := 2*x;
      V(m);
  }
]|
```

Example

- **Maintain *x≥0* in a program with arbitrary assignments to *x***

var *cv*: ***condition***;
m: ***binary semaphore*** *(initially := 1)*

```
Proc P1
|[
  while( true) {
      P(m);
      x := x-10;
      V(m);
  }
]|
```

```
Proc P2
|[
  while( true) {
      P(m);
      x := x + 50;
      x := 2*x;
      V(m);
  }
]|
```

Principle 1:

Where a condition **may be violated**:
check and **block**

Example

- **Maintain *x≥0* in a program with arbitrary assignments to *x***

```
var cv: condition;
      m: binary semaphore (initially := 1)
```

```
Proc P1
|[
  while( true) {
     P(m);
     while x<10  do
          V(m);
          Wait (cv);
          P(m) ;
     od;
     { x≥10 }
     x := x-10;
     V(m);
  }
]|
```

Ensure **mutual exclusion**

Check that *x≥0* after the execution of the CS

Allows others to check their condition on *x*

If the condition is false: Wait for a signal

Only one waiter at a time is allowed to: **re-check** and **execute the critical section**

Principle 1:

Where a condition **may be violated**: **check** and **block**

Example

- **Maintain $x \geq 0$ in a program with arbitrary assignments to *x***

```
var cv: condition;
    m: binary semaphore (initially := 1)
```

```
Proc P1
|[
  while( true) {
     P(m);
     while x<10  do
          V(m);
          Wait (cv);
          P(m) ;
     od;
     { x≥10 }
     x := x-10;
     V(m);
  }
]|
```

```
Proc P2
|[
  while( true) {
     P(m);
     x := x + 50;
     x := 2*x;
     V(m);
  }
]|
```

Principle 2:

Where a condition **may have become true**: **signal** the waiters

Example

- **Maintain *x≥0* in a program with arbitrary assignments to *x***

```
var cv: condition;
    m: binary semaphore (initially := 1)
```

```
Proc P1
|[
  while( true) {
     P(m);
     while x<10  do
          V(m);
          Wait (cv);
          P(m) ;
     od;
     { x≥10 }
     x := x-10;
     V(m);
  }
]|
```

```
Proc P2
|[
  while( true) {
     P(m);
     x := x + 50;
     x := 2*x;
     Sigall (cv);
     V(m);
  }
]|
```

Principle 2:

Where a condition **may have become true**: **signal** the waiters

Example

- **Maintain $x \geq 0$ in a program with arbitrary assignments to *x***

```
var cv: condition;
     m: binary semaphore (initially := 1)
```

```
Proc P1
|[
  while( true) {
      P(m);
      while x<10  do
             V(m);
             Wait (cv);
             P(m) ;
      od;
      { x≥10 }
      x := x-10;
      V(m);
  }
]|
```

Beware!! Not atomic!!!
Sigall(cv) may be **called before *Wait(cv)***
→ ***P1 is never woken up!***

```
Proc P2
|[
  while( true) {
      P(m);
      x := x + 50;
      x := 2*x;
      Sigall (cv);
      V(m);
  }
]|
```

Combine: condition variable & semaphore

- Need to combine "*V(m); Wait(cv)*" atomically:
- define *Wait(m,cv): <V(m); Wait(cv)>; P(m)*

```
var cv: condition;
    m: binary semaphore (initially := 1)
```

```
Proc P1
|[
  while( true) {
     P(m);
     while x<10  do
          Wait (m, cv);
     od;
     { x≥10 }
     x := x-10;
     V(m);
  }
]|
```

```
Proc P2
|[
  while( true) {
     P(m);
     x := x + 50;
     x := 2*x;
     Sigall (cv);
     V(m);
  }
]|
```

Using condition variables

- *cv* is **sometimes** extended with a **timeout mechanism**.
  - recheck the condition even though there is no signal.

- Each condition **variable *cv*** is **associated** with a **condition *B(cv)***
  - **Signaling** means *B(cv)* may have become true.
  - *B(cv)* can be a simple condition such as $(x \geq 10)$…
  - **…or a combined condition** such as $B(cv)=(x \geq 10 \lor y \geq 5 \lor x+y \geq 12)$
    - **alternatively**, use **3 condition vars** $B(cv1)=(x \geq 10)$, $B(cv2)=(y \geq 5)$, $B(cv3)=(x+y \geq 12)$
    - …using a **single semaphore** associated with all critical sections (e.g. POSIX).

- Depending on the context, a **single or multiple** condition **variables** is used **for all conditions**
  - using more than one is an efficiency choice
  - For example: JAVA allows a single implicit condition variable per object (actually, JAVA provides a *Monitor* mechanism)

POSIX: condition variables (1003.1c)

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
status = pthread_cond_init (&cond, attr);               /* should return 0              */
status = pthread_cond_destroy (&cond);                  /* idem                         */
status = pthread_cond_wait (&cond, m);                  /* semaphore m is associated with
                                                         * all critical sections        */
status = pthread_cond_timedwait (&cond, m, exp);        /*  exp: max. waiting time; returns
                                                         * ETIMEDOUT after exp.*/
status = pthread_cond_signal (&cond);                   /* signal one waiter            */
status = pthread_cond_broadcast (&cond);                /* signal all waiters           */
```